Context-Free Grammars
A Motivating Question
```python
>>> (137 + 42) - 2 * 3
173
```
```python
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
173
```
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
137

>>>
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
137

>>> (200 / 2) + 6 / 2
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
137

>>> (200 / 2) + 6 / 2
103.0

>>>
Mad Libs for Arithmetic

( __Int__ __Op__ __Int__ )

( __Op__ __Int__ __Op__ __Int__ )
Mad Libs for Arithmetic

( \frac{26}{\text{Int}} + \frac{42}{\text{Int}} ) \times \frac{2}{\text{Int}} + \frac{1}{\text{Int}}

Slide credit: Amy Liu
Mad Libs for Arithmetic

( __ Int __ Op __ Int __ Op __ Int __ Op __ Int __ )
Mad Libs for Arithmetic

( \frac{7}{\text{Int} \quad \text{Op}} \frac{5}{\text{Int}} ) \div \frac{5}{\text{Int} \quad \text{Op}} \frac{49}{\text{Int}}
Mad Libs for Arithmetic

This only lets us make arithmetic expressions of the form \((\text{Int Op Int}) \text{ Op Int Op Int}\).

What about arithmetic expressions that don’t follow this pattern?
Recursive Mad Libs

Expr
Recursive Mad Libs

What can an arithmetic expression be?
What can an arithmetic expression be?

int  A single number.
Recursive Mad Libs

What can an arithmetic expression be?

\textbf{int} \quad A single number.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
  - A single number.
- **Expr Op Expr**
  - Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

int  A single number.

Expr Op Expr  Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

\[
\text{int} \quad \text{Expr} \quad \text{Op} \quad \text{Expr}
\]

- A single number.
- Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
- **+**
- **Expr**
- **Op**
- **Expr**

A single number.

Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
- **Expr**
- **Op**
- **Expr**

A single number.

Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

- `int` A single number.
- `Expr Op Expr` Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

\[
\begin{array}{c}
\text{int} \\
\text{Expr} \\
\text{Op} \\
\text{Expr} \\
\text{Op} \\
\text{Expr}
\end{array}
\]

- A single number.
- Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

\[
\text{int} \quad \text{+} \quad \text{int}
\]

\[
\text{Expr} \quad \text{Op} \quad \text{Expr} \quad \text{Op} \quad \text{Expr}
\]

A single number.

Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

**int**

A single number.

**Expr Op Expr**

Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
- **+**
- **int**
- **×**
- **int**

- **Expr** **Op** **Expr** **Op** **Expr**

- A single number.
- Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

`int` A single number.

`Expr Op Expr` Two expressions joined by an operator.
Recursive Mad Libs

What can an arithmetic expression be?

<table>
<thead>
<tr>
<th>Expr</th>
<th>A single number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>A parenthesesized expression.</td>
</tr>
<tr>
<td>Expr Op Expr (Expr)</td>
<td>Two expressions joined by an operator.</td>
</tr>
</tbody>
</table>
Recursive Mad Libs

What can an arithmetic expression be?

- `int` : A single number.
- `Expr Op Expr` : Two expressions joined by an operator.
- `(Expr)` : A parenthesized expression.
What can an arithmetic expression be?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>A single number.</td>
</tr>
<tr>
<td>Expr Op Expr</td>
<td>Two expressions joined by an operator.</td>
</tr>
<tr>
<td>(Expr)</td>
<td>A parenthesized expression.</td>
</tr>
</tbody>
</table>
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
  - A single number.
- **Expr Op Expr**
  - Two expressions joined by an operator.
- **(Expr)**
  - A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
  - A single number.

- **Expr Op Expr**
  - Two expressions joined by an operator.

- **(Expr)**
  - A parenthensized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**  
  A single number.

- **Expr Op Expr**  
  Two expressions joined by an operator.

- **(Expr)**  
  A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
- **Expr Op Expr**
- **(Expr)**

A single number.
Two expressions joined by an operator.
A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

<table>
<thead>
<tr>
<th>int</th>
<th>A single number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr Op Expr</td>
<td>Two expressions joined by an operator.</td>
</tr>
<tr>
<td>(Expr)</td>
<td>A parenthesized expression.</td>
</tr>
</tbody>
</table>
What can an arithmetic expression be?

- **Int**  
  A single number.

- **Expr Op Expr**  
  Two expressions joined by an operator.

- **(Expr)**  
  A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
  - A single number.

- **Expr Op Expr**
  - Two expressions joined by an operator.

- **(Expr)**
  - A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- **int**
  - A single number.
- **Expr Op Expr**
  - Two expressions joined by an operator.
- **(Expr)**
  - A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- int
- Expr Op Expr
- (Expr)

A single number.
Two expressions joined by an operator.
A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- `int` A single number.
- `Expr Op Expr` Two expressions joined by an operator.
- `(Expr)` A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- int
- Expr Op Expr
- (Expr)

- int
  A single number.

- Expr Op Expr
  Two expressions joined by an operator.

- (Expr)
  A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

- `int` A single number.
- `Expr Op Expr` Two expressions joined by an operator.
- `(Expr)` A parenthesized expression.
Recursive Mad Libs

What can an arithmetic expression be?

*int* A single number.

*Expr Op Expr* Two expressions joined by an operator.

*(Expr)* A parenthesized expression.
A context-free grammar (or CFG) is a recursive set of rules that define a language.

(There’s a bunch of specific requirements about what those rules can be; more on that in a bit.)
Arithmetic Expressions

Here’s how we might express the recursive rules from earlier as a CFG.

```
Expr → int
Expr → Expr Op Expr
Expr → (Expr)
Op → +
Op → −
Op → ×
Op → /
```

This is called a production rule. It says “if you see Expr, you can replace it with Expr Op Expr.”
Arithmetic Expressions

Here’s how we might express the recursive rules from earlier as a CFG.

```
Expr → int
Expr → Expr Op Expr
Expr → (Expr)
Op → +
Op → –
Op → ×
Op → /
```

This one says “if you see Op, you can replace it with –.”
Arithmetic Expressions

- Here’s how we might express the recursive rules from earlier as a CFG.

```
Expr → int
Expr → Expr Op Expr
Expr → (Expr)
Op → +
Op → −
Op → ×
Op → /
```
Arithmetic Expressions

Here’s how we might express the recursive rules from earlier as a CFG.

\[
\begin{align*}
\text{Expr} & \rightarrow \text{int} \\
\text{Expr} & \rightarrow \text{Expr} \ \text{Op} \ \text{Expr} \\
\text{Expr} & \rightarrow (\text{Expr}) \\
\text{Op} & \rightarrow + \\
\text{Op} & \rightarrow - \\
\text{Op} & \rightarrow \times \\
\text{Op} & \rightarrow / \\
\end{align*}
\]

These red symbols are called \textit{nonterminals}. They’re placeholders that get expanded later on.
Arithmetic Expressions

Here’s how we might express the recursive rules from earlier as a CFG.

\[
\begin{align*}
\text{Expr} & \rightarrow \text{int} \\
\text{Expr} & \rightarrow \text{Expr Op Expr} \\
\text{Expr} & \rightarrow (\text{Expr}) \\
\text{Op} & \rightarrow + \\
\text{Op} & \rightarrow - \\
\text{Op} & \rightarrow \times \\
\text{Op} & \rightarrow / \\
\end{align*}
\]

The symbols in blue monospace are **terminals**. They’re the final characters used in the string and never get replaced.
Arithmetic Expressions

Here’s how we might express the recursive rules from earlier as a CFG.

```
Expr → int
Expr → Expr Op Expr
Expr → (Expr)
Op → +
Op → −
Op → ×
Op → /
```

```
Expr
⇒ Expr Op Expr
⇒ Expr Op (Expr)
⇒ Expr Op (Expr Op Expr)
⇒ Expr × (Expr Op Expr)
⇒ int × (Expr Op Expr)
⇒ int × (int Op Expr)
⇒ int × (int Op int)
⇒ int × (int + int)
```
Context-Free Grammars

• Formally, a context-free grammar is a collection of four items:
  • a set of *nonterminal symbols* (also called *variables*),
  • a set of *terminal symbols* (the *alphabet* of the CFG),
  • a set of *production rules* saying how each nonterminal can be replaced by a string of terminals and nonterminals, and
  • a *start symbol* (which must be a nonterminal) that begins the derivation. By convention, the start symbol is the one on the left-hand side of the first production.

\[
\begin{align*}
\text{Expr} & \rightarrow \text{int} \\
\text{Expr} & \rightarrow \text{Expr} \ \text{Op} \ \text{Expr} \\
\text{Expr} & \rightarrow (\text{Expr}) \\
\text{Op} & \rightarrow + \\
\text{Op} & \rightarrow - \\
\text{Op} & \rightarrow \times \\
\text{Op} & \rightarrow / 
\end{align*}
\]
Some CFG Notation

• In today’s slides, capital letters in **Bold Red Uppercase** will represent nonterminals.
  • e.g. A, B, C, D

• Lowercase letters in **blue monospace** will represent terminals.
  • e.g. t, u, v, w

• Lowercase Greek letters in **gray italics** will represent arbitrary strings of terminals and nonterminals.
  • e.g. α, γ, ω

• You don't need to use these conventions on your own; just make sure whatever you do is readable.
A Notational Shorthand

\[
\begin{align*}
\text{Expr} & \rightarrow \text{int} \\
\text{Expr} & \rightarrow \text{Expr} \text{ Op} \text{ Expr} \\
\text{Expr} & \rightarrow (\text{Expr}) \\
\text{Op} & \rightarrow + \\
\text{Op} & \rightarrow - \\
\text{Op} & \rightarrow \times \\
\text{Op} & \rightarrow / 
\end{align*}
\]
A Notational Shorthand

\[
\begin{align*}
\text{Expr} & \rightarrow \text{int} \mid \text{Expr Op Expr} \mid (\text{Expr}) \\
\text{Op} & \rightarrow + \mid - \mid \times \mid / 
\end{align*}
\]
Derivations

- A sequence of zero or more steps where nonterminals are replaced by the right-hand side of a production is called a **derivation**.

- If string $\alpha$ derives string $\omega$, we write $\alpha \Rightarrow^* \omega$.

- In the example on the left, we see that $\text{Expr} \Rightarrow^* \text{int} \times (\text{int} + \text{int})$.

```
Expr → int | Expr Op Expr | (Expr)
Op → + | - | × | /
```
The Language of a Grammar

- If $G$ is a CFG with alphabet $\Sigma$ and start symbol $S$, then the \textit{language of $G$} is the set

\[ \mathcal{L}(G) = \{ \omega \in \Sigma^* \mid S \Rightarrow^* \omega \} \]

- That is, $\mathcal{L}(G)$ is the set of strings of terminals derivable from the start symbol.
If $G$ is a CFG with alphabet $\Sigma$ and start symbol $S$, then the *language of $G$* is the set

$$\mathcal{L}(G) = \{ \omega \in \Sigma^* | S \Rightarrow^* \omega \}$$

Consider the following CFG $G$ over $\Sigma = \{a, b, c, d\}$:

$$S \rightarrow Sa \mid dT$$

$$T \rightarrow bTb \mid c$$

Which of the following strings are in $\mathcal{L}(G)$?

- dca
- dc
- cad
- bcb
- dTaa

*Answer at pollev.com/zhenglian740*
Context-Free Languages

• A language $L$ is called a \textit{context-free language} (or CFL) if there is a CFG $G$ such that $L = \mathcal{L}(G)$.

• Questions:
  • How are context-free and regular languages related?
  • How do we design context-free grammars for context-free languages?
Context-Free Languages

A language $L$ is called a \textit{context-free language} (or CFL) if there is a CFG $G$ such that $L = \mathcal{L}(G)$.

Questions:

- How are context-free and regular languages related?

How do we design context-free grammars for context-free languages?
Five Possibilities
CFGs and Regular Expressions

- CFGs consist purely of production rules of the form $A \rightarrow \omega$. They do not have the regular expression operators $*$ or $\cup$.

- You can use the symbols $*$ and $\cup$ if you’d like in a CFG, but they just stand for themselves.

- Consider this CFG $G$:

  $$S \rightarrow a^*b$$

- Here, $\mathcal{L}(G) = \{a^*b\}$ and has cardinality one. That is, $\mathcal{L}(G) \neq \{a^n b \mid n \in \mathbb{N}\}$. 
CFGs and Regular Expressions

• **Theorem:** Every regular language is context-free.

• **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[ a ( b \cup \varepsilon ) c \]
CFGs and Regular Expressions

- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.
CFGs and Regular Expressions

• **Theorem:** Every regular language is context-free.
• **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[ a ( b \cup \varepsilon ) c \]
CFGs and Regular Expressions

- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[
S \rightarrow aXc
\]

\[
\text{a ( b } \cup \varepsilon \text{ ) c}
\]
CFGs and Regular Expressions

- **Theorem**: Every regular language is context-free.
- **Proof idea**: Show how to convert an arbitrary regular expression into a context-free grammar.

\[
\begin{align*}
S & \rightarrow aXc \\
X & \rightarrow b \mid \epsilon \\
\end{align*}
\]
CFGs and Regular Expressions

- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

It’s totally fine for a production to replace a nonterminal with the empty string.
CFGs and Regular Expressions

• **Theorem:** Every regular language is context-free.

• **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[(a \cup b)^2 c^*\]
CFGs and Regular Expressions

- **Theorem**: Every regular language is context-free.
- **Proof idea**: Show how to convert an arbitrary regular expression into a context-free grammar.
**Theorem:** Every regular language is context-free.

**Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.
CFGs and Regular Expressions

• **Theorem:** Every regular language is context-free.
• **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[
S \rightarrow XY \\
(a \cup b)^2 c^* \\
X \\
Y
\]
CFGs and Regular Expressions

• **Theorem:** Every regular language is context-free.

• **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[ S \rightarrow XY \]

\[ (a \cup b)^2 c^* \]
CFGs and Regular Expressions

- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[
S \rightarrow XY
\]

The diagram shows a context-free grammar for the regular expression \((a \cup b)^2c^*\).
CFGs and Regular Expressions

- **Theorem**: Every regular language is context-free.
- **Proof idea**: Show how to convert an arbitrary regular expression into a context-free grammar.

\[
\begin{align*}
S & \rightarrow XY \\
X & \rightarrow ZZ \\
(a \cup b)^2 c^* & \\
\end{align*}
\]
CFGs and Regular Expressions

- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[
\begin{align*}
S &\rightarrow XY \\
X &\rightarrow ZZ \\
Z &\rightarrow a \mid b
\end{align*}
\]
CFGs and Regular Expressions

• **Theorem:** Every regular language is context-free.

• **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

\[
\begin{align*}
S &\rightarrow XY \\
X &\rightarrow ZZ \\
Z &\rightarrow a \mid b \\
Y &\rightarrow cY \mid \varepsilon \\
(a \cup b)^2 &\rightarrow Z \\
&\rightarrow X \\
&\rightarrow Y
\end{align*}
\]
Two Five Possibilities

- REG
- CFL

- REG
- CFL

- REG
- CFL

- REG
- CFL

- REG
- CFL
Two Possibilities
The Language of a Grammar

- Consider the following CFG $G$:
  \[
  S \rightarrow aSb \mid \varepsilon
  \]

- What strings can this generate?
The Language of a Grammar

• Consider the following CFG $G$:
  \[ S \rightarrow aSb \mid \varepsilon \]

• What strings can this generate?
The Language of a Grammar

- Consider the following CFG $G$:
  \[ S \rightarrow aSb \mid \varepsilon \]

- What strings can this generate?

\[ aSb \]
The Language of a Grammar

- Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

- What strings can this generate?
The Language of a Grammar

- Consider the following CFG $G$:

  $S \rightarrow aSb \mid \varepsilon$

- What strings can this generate?

```
 a a S b b b
```
The Language of a Grammar

- Consider the following CFG $G$:
  \[ S \rightarrow aSb \mid \varepsilon \]

- What strings can this generate?

  \[ a \ a \quad S \quad b \ b \]
The Language of a Grammar

• Consider the following CFG $G$:
  $$S \rightarrow aSb \mid \varepsilon$$

• What strings can this generate?

| a | a | a | S | b | b | b | b | b |
The Language of a Grammar

- Consider the following CFG $G$:
  $$ S \rightarrow aSb \mid \varepsilon $$

- What strings can this generate?

  $a a a S b b b b$
The Language of a Grammar

• Consider the following CFG $G$:
  \[ S \rightarrow aSb \mid \varepsilon \]

• What strings can this generate?

\[ a \ a \ a \ a \ a \ S \ b \ b \ b \ b \ b \ b \]
The Language of a Grammar

- Consider the following CFG $G$:
  
  $S \rightarrow aSb \mid \varepsilon$

- What strings can this generate?

| a | a | a | a | a | b | b | b | b | b | b |
The Language of a Grammar

• Consider the following CFG $G$:
  
  \[ S \rightarrow aSb \mid \varepsilon \]

• What strings can this generate?

\[
\begin{array}{cccccccccc}
  a & a & a & a & b & b & b & b & b & b \\
\end{array}
\]
The Language of a Grammar

- Consider the following CFG $G$:

$$S \rightarrow aSb \mid \varepsilon$$

- What strings can this generate?

$$\mathcal{L}(G) = \{ a^n b^n \mid n \in \mathbb{N} \}$$
Regular Languages ⊂ CFLs ⊂ All Languages
Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- **Intuition:** Derivations of strings have unbounded “memory.”
Why the Extra Power?

• Why do CFGs have more power than regular expressions?

• *Intuition:* Derivations of strings have unbounded “memory.”

\[ S \rightarrow aSb \mid \varepsilon \]
Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- **Intuition:** Derivations of strings have unbounded “memory.”

  \[ S \rightarrow aSb \mid \varepsilon \]
Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- **Intuition:** Derivations of strings have unbounded “memory.”

\[ S \rightarrow aSb \mid \varepsilon \]
Why the Extra Power?

• Why do CFGs have more power than regular expressions?

• **Intuition:** Derivations of strings have unbounded “memory.”

\[
S \rightarrow aSb \mid \varepsilon
\]

a  S  b
Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- **Intuition:** Derivations of strings have unbounded "memory."

\[
S \rightarrow aSb \mid \varepsilon
\]

```
| a | a | S | b | b |
```
Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- **Intuition:** Derivations of strings have unbounded “memory.”

\[ S \rightarrow aSa | \varepsilon \]

- **Example:** Derivation of the string `aaSbb`:
  - **Step 1:** `S` expands to `aSa`.
  - **Step 2:** The `S` in the middle expands to `bb`, resulting in `aSbb`.
  - **Result:** The final string is `aaSbb`.

This example illustrates how CFGs can derive strings that regular expressions cannot.
Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- **Intuition:** Derivations of strings have unbounded “memory.”

\[ S \rightarrow aSb \mid \varepsilon \]

```
  a  a  a  S  b  b  b  b
```
Why the Extra Power?

• Why do CFGs have more power than regular expressions?

• *Intuition:* Derivations of strings have unbounded “memory.”

\[ S \rightarrow aSb \mid \varepsilon \]

---

```
  a  a  a  S  b  b  b  b
```
Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- **Intuition:** Derivations of strings have unbounded “memory.”

\[
S \rightarrow aSb \mid \varepsilon
\]

\[
\begin{array}{cccccccc}
a & a & a & a & a & S & b & b & b & b & b
\end{array}
\]
Why the Extra Power?

- Why do CFGs have more power than regular expressions?

  **Intuition:** Derivations of strings have unbounded “memory.”

  \[
  S \to aSb \mid \varepsilon
  \]

  
  a a a a a b b b b b
Why the Extra Power?

• Why do CFGs have more power than regular expressions?

• *Intuition:* Derivations of strings have unbounded “memory.”

\[ S \rightarrow aSb \mid \varepsilon \]

```
  a  a  a  a  a  b  b  b  b  b  b
```
Time-Out for Announcements!
Problem Set Six

• Problem Set Five was due today at 5:30PM.

• Problem Set Six goes out today. It’s due next Friday at 5:30PM.
  • It’s all about regular expressions, properties of regular languages, and nonregular languages.
Preparing for the Final Exam

• We’ve released two practice final exams. We strongly recommend sitting down and taking the practice exam under realistic exam conditions.

• There is also a gigantic compendium of CS103 practice problems on the course website.
  • You can search for problems based on the topics they cover, whether solutions are available, whether they’re ones we particularly like, and whether they have solutions.
  • Please do not read the solutions to a problem until you have worked through it.
Back to CS103!
Designing CFGs

• Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft.

• When thinking about CFGs:
  
  • *Think recursively:* Build up bigger structures from smaller ones.
  
  • *Have a construction plan:* Know in what order you will build up the string.
  
  • *Store information in nonterminals:* Have each nonterminal correspond to some useful piece of information.
Designing CFGs

• Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid w$ is a palindrome $\}$

• We can design a CFG for $L$ by thinking inductively:
  • Base case: $\varepsilon$, $a$, and $b$ are palindromes.
  • If $\omega$ is a palindrome, then $a\omega a$ and $b\omega b$ are palindromes.
  • No other strings are palindromes.

\[
S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb
\]
Designing CFGs

- Let $\Sigma = \{\{, \}\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Some sample strings in $L$:

```
{{{{}}}}
{{}}{}
{{}}}{}
{{}}}{{}}
{{}}}{{}}
{{}}}{{}}
{{}}}{{}}
ε
{{}}{{}}
```
Designing CFGs

- Let $\Sigma = \{\{, \}\}$ and let $L = \{ w \in \Sigma^* \mid w$ is a string of balanced braces $\}$

- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.
Designing CFGs

- Let $\Sigma = \{\{,\}\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.
Designing CFGs

• Let $\Sigma = \{\{, \}\}$ and let $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced braces }\}$

• Let's think about this recursively.
  • Base case: the empty string is a string of balanced braces.
  • Recursive step: Look at the closing brace that matches the first open brace.
Designing CFGs

- Let $\Sigma = \{\{, \}\}$ and let $L = \{ w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.
Designing CFGs

• Let $\Sigma = \{\{, \}\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces $\}$

• Let's think about this recursively.
  • Base case: the empty string is a string of balanced braces.
  • Recursive step: Look at the closing brace that matches the first open brace. Removing the first brace and the matching brace forms two new strings of balanced braces.

$$S \rightarrow \{S\}S \mid \varepsilon$$
Designing CFGs

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid w$ has the same number of a's and b's $\}$

How many of the following CFGs have language $L$?

- $S \rightarrow aSb \mid bSa \mid \varepsilon$
- $S \rightarrow abS \mid baS \mid \varepsilon$
- $S \rightarrow abSba \mid baSab \mid \varepsilon$
- $S \rightarrow SbaS \mid SabS \mid \varepsilon$

*Answer at pollev.com/cs103*
Designing CFGs

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* | w$ has the same number of $a$'s and $b$'s $\}$

How many of the following CFGs have language $L$?

- $S \rightarrow aSb \mid bSa \mid \varepsilon$
- $S \rightarrow abS \mid baS \mid \varepsilon$
- $S \rightarrow abSba \mid baSab \mid \varepsilon$
- $S \rightarrow SbaS \mid SabS \mid \varepsilon$
Designing CFGs

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid w$ has the same number of a's and b's $\}$

How many of the following CFGs have language $L$?

- $S \rightarrow aSb \mid bSa \mid \varepsilon$
- $S \rightarrow abS \mid baS \mid \varepsilon$
- $S \rightarrow abSba \mid baS\text{ab} \mid \varepsilon$
- $S \rightarrow Sbas \mid S\text{ab}S \mid \varepsilon$
Designing CFGs

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid w$ has the same number of $a$'s and $b$'s $\}$

How many of the following CFGs have language $L$?

- $S \rightarrow aSb \mid bSa \mid \varepsilon$
- $S \rightarrow abS \mid baS \mid \varepsilon$
- $S \rightarrow abSba \mid baSab \mid \varepsilon$
- $S \rightarrow SbS \mid SabS \mid \varepsilon$
Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid w$ has the same number of $a$'s and $b$'s $\}$

How many of the following CFGs have language $L$?

- $S \rightarrow aSb \mid bSa \mid \varepsilon$
- $S \rightarrow abS \mid baS \mid \varepsilon$
- $S \rightarrow abSba \mid baSab \mid \varepsilon$
- $S \rightarrow SbaS \mid SabS \mid \varepsilon$
Designing CFGs: A Caveat

• When designing a CFG for a language, make sure that it
  • generates all the strings in the language and
  • never generates a string outside the language.

• The first of these can be tricky – make sure to test your grammars!

• You'll design your own CFG for this language on Problem Set 8.
CFG Caveats II

• Is the following grammar a CFG for the language \{ a^n b^n \mid n \in \mathbb{N} \}?  
  \[ S \rightarrow aSb \]

• What strings in \{a, b\}^* can you derive?  
  • Answer: \textit{None!}

• What is the language of the grammar?  
  • Answer: \( \emptyset \)

• When designing CFGs, make sure your recursion actually terminates!
Designing CFGs

- When designing CFGs, remember that each nonterminal can be expanded out independently of the others.

- Let $\Sigma = \{a, \varepsilon\}$ and let $L = \{a^n \varepsilon a^n \mid n \in \mathbb{N}\}$.

- Is the following a CFG for $L$?

  $S \rightarrow X \varepsilon X$
  $X \rightarrow aX \mid \varepsilon$

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \varepsilon X$</td>
</tr>
<tr>
<td>$aX \varepsilon X$</td>
</tr>
<tr>
<td>$aaX \varepsilon X$</td>
</tr>
<tr>
<td>$aa \varepsilon X$</td>
</tr>
<tr>
<td>$aa \varepsilon aX$</td>
</tr>
<tr>
<td>$aa \varepsilon a$</td>
</tr>
</tbody>
</table>
Finding a Build Order

- Let $\Sigma = \{a, \frac{a}{a}\}$ and let $L = \{a^n\frac{a}{a}a^n \mid n \in \mathbb{N}\}$.
- To build a CFG for $L$, we need to be more clever with how we construct the string.
  - If we build the strings of $a$'s independently of one another, then we can't enforce that they have the same length.
  - **Idea:** Build both strings of $a$'s at the same time.
- Here's one possible grammar based on that idea:
  
  $S \rightarrow \frac{a}{a} | aSa$

  $\Rightarrow aSa$
  $\Rightarrow aaSaa$
  $\Rightarrow aaaSaaa$
  $\Rightarrow aaa\frac{a}{a}aaa$
Storing Information in Nonterminals

- **Key idea:** Different non-terminals should represent different states or different types of strings.
  - For example, different phases of the build, or different possible structures for the string.
  - Think like the same ideas from DFA/NFA design where states in your automata represent pieces of information.
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 

- Examples:

  $\varepsilon \in L$ 
  $a \notin L$
  $abb \in L$ 
  $b \notin L$
  $bab \in L$ 
  $ababab \notin L$
  $aababa \in L$ 
  $aabaaaaaa \notin L$
  $bbbbbbb \in L$ 
  $bbbb \notin L$
Storing Information in Nonterminals

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv 3 \pmod{2}$ and all the characters in the first third of $w$ are the same $\}$.

Examples:

- $\varepsilon \in L$  
- $a \not\in L$
- $a \ v  b \in L$
- $b \not\in L$
- $b \ v  a \ b \in L$
- $a b \ a b a b \not\in L$
- $a b a b a b a b a b a b a b \not\in L$
- $b b \ b b b b \in L$
- $b b b b \not\in L$
- $b b b b \not\in L$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $.\}$.
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 

- One approach:

<table>
<thead>
<tr>
<th>aaa</th>
<th>bab</th>
<th>Observation 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>abb</td>
<td>bbb</td>
<td>Strings in this</td>
</tr>
<tr>
<td>aaabab</td>
<td>bbabbb</td>
<td>language are either:</td>
</tr>
<tr>
<td>aababa</td>
<td>bbbaaaaaa</td>
<td>the first third is $a$s or</td>
</tr>
<tr>
<td>aaaaaaaa</td>
<td>bbbbbbabaa</td>
<td>the first third is $b$s.</td>
</tr>
</tbody>
</table>
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0 \text{ and all the characters in the first third of } w \text{ are the same}\}.

- One approach:

  
  
  \[
  \begin{align*}
  \text{aaa} & \quad \text{bab} \\
  \text{abb} & \quad \text{bbb} \\
  \text{aaabab} & \quad \text{bbabbb} \\
  \text{aababa} & \quad \text{bbbaaaaaaa} \\
  \text{aaaaaaaaaa} & \quad \text{bbbbbbabaa}
  \end{align*}
  \]
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $.$

- One approach:

  - Observation 2: Amongst these strings, for every $a$ I have in the first third, I need two other characters in the last two thirds.
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv 3 \text{ and all the characters in the first third of } w \text{ are the same}\}$.

- One approach:

  aaa  bab
  abb  bbb
  aaabab bbabbb
  aaaaaaaaaa bababaa

  This pattern of “for every x I see here, I need a y somewhere else in the string” is very common in CFGs!

**Observation 2:**

Amongst these strings, for every $a$ I have in the first third, I need two other characters in the last two thirds.
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.

- One approach:

  $\begin{align*}
  \text{aaa} & \quad \text{bab} \\
  \text{abb} & \quad \text{bbb} \\
  \text{aaabab} & \quad \text{bbabbb} \\
  \text{aababa} & \quad \text{bbbaaaaaa} \\
  \text{aaaaaaaaaa} & \quad \text{bbbbbbabaa}
  \end{align*}$

  **Observation 2:**

  Amongst these strings, for every $a$ I have in the first third, I need two other characters in the last two thirds.
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv 3 \ 0$ and all the characters in the first third of $w$ are the same $\}$.
- One approach:

\[
\begin{align*}
\text{aaa} \\
\text{abb} \\
\text{aaabab} \\
\text{aababa} \\
\text{aaaaaaa} \\
\text{A} &\rightarrow aAXX \mid \varepsilon \\
\text{X} &\rightarrow a \mid b
\end{align*}
\]

Here the nonterminal $A$ represents “a string where the first third is $a$’s” and the nonterminal $X$ represents “any character”
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$.

- One approach:

  - $aaa$  $bab$
  - $abb$  $bbb$
  - $aaabab$  $bbabbb$
  - $aababa$  $bbbaaaaaa$
  - $aaaaaaaaaa$  $bbbbbabaaa$

  $A \rightarrow aAXX \mid \varepsilon$  $X \rightarrow a \mid b$
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 

- One approach:

  \[
  \begin{align*}
  \text{aaa} & \quad \text{bab} \\
  \text{abb} & \quad \text{bbb} \\
  \text{aaabab} & \quad \text{bbabbb} \\
  \text{aababa} & \quad \text{bbbaaaaaa} \\
  \text{aaaaaaaa} & \quad \text{bbbbbbbb} \\
  \end{align*}
  \]

  $B \rightarrow bBX \mid \varepsilon \quad X \rightarrow a \mid b$
Storing Information in Nonterminals

• Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* | |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 

• Tying everything together:

$S \rightarrow A \mid B$

$A \rightarrow aAXX \mid \varepsilon$

$B \rightarrow bBXX \mid \varepsilon$

$X \rightarrow a \mid b$
Storing Information in Nonterminals

• Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of $w$ are the same $\}$. 

• Tying everything together:

$$
S \rightarrow A \mid B \\
A \rightarrow aAXX \mid \varepsilon \\
B \rightarrow bBXX \mid \varepsilon \\
X \rightarrow a \mid b
$$

Overall strings in this language either follow the pattern of $A$ or $B$. 
Storing Information in Nonterminals

Let Σ = \{a, b\} and let \( L = \{ w \in \Sigma^* \mid |w| \equiv_{3} 0 \) and all the characters in the first third of \( w \) are the same \}.

- Tying everything together:

\[
S \rightarrow A \mid B \\
A \rightarrow aAXX \mid \varepsilon \\
B \rightarrow bBXX \mid \varepsilon \\
X \rightarrow a \mid b
\]

\( A \) represents “strings where the first third is \( a \)’s”
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv 3 \ 0$ and all the characters in the first third of $w$ are the same $\}$. 

- Tying everything together:

$$
S \rightarrow A \mid B \\
A \rightarrow aAXX \mid \varepsilon \\
B \rightarrow bBXX \mid \varepsilon \\
X \rightarrow a \mid b
$$

$B$ represents “strings where the first third is $b$’s”
Storing Information in Nonterminals

- Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv 3 \text{ mod } 0 \text{ and all the characters in the first third of } w \text{ are the same}\}$.  
- Tying everything together:

  $$
  \begin{align*}
  S & \rightarrow A \mid B \\
  A & \rightarrow aAXX \mid \varepsilon \\
  B & \rightarrow bBXX \mid \varepsilon \\
  X & \rightarrow a \mid b
  \end{align*}
  $$

  $X$ represents “either an $a$ or a $b$”
Function Prototypes

- Let $\Sigma = \{\text{void, int, double, name, (, ), , , ;}\}$. 
- Let's write a CFG for C-style function prototypes!
- Examples:
  - \text{void name(int name, double name)};
  - \text{int name()};
  - \text{int name(double name)};
  - \text{int name(int, int name, int)};
  - \text{void name(void)};
Function Prototypes

• Here's one possible grammar:
  
  • $S \rightarrow \text{Ret name (Args)}$
  
  • $\text{Ret} \rightarrow \text{Type} \mid \text{void}$
  
  • $\text{Type} \rightarrow \text{int} \mid \text{double}$
  
  • $\text{Args} \rightarrow \varepsilon \mid \text{void} \mid \text{ArgList}$
  
  • $\text{ArgList} \rightarrow \text{OneArg} \mid \text{ArgList}, \text{OneArg}$
  
  • $\text{OneArg} \rightarrow \text{Type} \mid \text{Type name}$
  
• Fun question to think about: what changes would you need to make to support pointer types?
Summary of CFG Design Tips

• Look for recursive structures where they exist: they can help guide you toward a solution.

• Keep the build order in mind – often, you'll build two totally different parts of the string concurrently.
  • Usually, those parts are built in opposite directions: one's built left-to-right, the other right-to-left.

• Use different nonterminals to represent different structures.
Applications of Context-Free Grammars
CFGs for Programming Languages

\[
\begin{align*}
\text{BLOCK} & \rightarrow \text{STMT} \\
& \quad | \{ \text{STMTS} \} \\
\text{STMTS} & \rightarrow \epsilon \\
& \quad | \text{STMT} \text{STMTS} \\
\text{STMT} & \rightarrow \text{EXPR;} \\
& \quad | \text{if (EXPR) BLOCK} \\
& \quad | \text{while (EXPR) BLOCK} \\
& \quad | \text{do BLOCK while (EXPR);} \\
& \quad | \text{BLOCK} \\
& \quad | \ldots \\
\text{EXPR} & \rightarrow \text{identifier} \\
& \quad | \text{constant} \\
& \quad | \text{EXPR + EXPR} \\
& \quad | \text{EXPR - EXPR} \\
& \quad | \text{EXPR * EXPR} \\
& \quad | \ldots
\end{align*}
\]
Grammars in Compilers

• One of the key steps in a compiler is figuring out what a program “means.”

• This is usually done by defining a grammar showing the high-level structure of a programming language.

• There are certain classes of grammars (LL(1) grammars, LR(1) grammars, LALR(1) grammars, etc.) for which it's easy to figure out how a particular string was derived.

• Tools like yacc or bison automatically generate parsers from these grammars.

• Curious to learn more? Take CS143!
Natural Language Processing

• By building context-free grammars for actual languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence.

  • In fact, CFGs were first called phrase-structure grammars and were introduced by Noam Chomsky in his seminal work *Syntactic Structures*.

  • They were then adapted for use in the context of programming languages, where they were called Backus-Naur forms.

• The Stanford Parser project is one place to look for an example of this.

• Want to learn more? Take CS124 or CS224N!
Next Time

- **Turing Machines**
  - What does a computer with unbounded memory look like?
  - How would you program it?
  - What can you do with it?